

THE ISOMORPHISM PROBLEM FOR PLANAR 3-CONNECTED GRAPHS IS IN UNAMBIGUOUS LOGSPACE

THOMAS THIERAUF¹ AND FABIAN WAGNER²

¹ Fak. Elektronik und Informatik, HTW Aalen, 73430 Aalen, Germany
E-mail address: `thomas.thierauf@uni-ulm.de`

² Inst. für Theoretische Informatik, Universität Ulm, 89069 Ulm, Germany
E-mail address: `fabian.wagner@uni-ulm.de`

ABSTRACT. The isomorphism problem for planar graphs is known to be efficiently solvable. For planar 3-connected graphs, the isomorphism problem can be solved by efficient parallel algorithms, it is in the class \mathbf{AC}^1 .

In this paper we improve the upper bound for planar 3-connected graphs to unambiguous logspace, in fact to $\mathbf{UL} \cap \mathbf{coUL}$. As a consequence of our method we get that the isomorphism problem for oriented graphs is in \mathbf{NL} . We also show that the problems are hard for \mathbf{L} .

1. Introduction

The graph isomorphism problem (GI) is one of the most challenging problems today. No polynomial time algorithm is known for it, even with extended resources like randomization or on quantum computers. On the other hand, it is not known to be \mathbf{NP} -complete and there are good reasons to conjecture that it is in fact not complete.

For some restricted classes of graphs, efficient algorithms for GI are known. For example for trees [AHU74] or for graphs with bounded degree [Luk82]. We are interested in planar graphs and 3-connected graphs. A graph is 3-connected if it remains connected after deleting two arbitrary vertices. In 1966, Weinberg [Wei66] presented an $O(n^2)$ -algorithm for testing isomorphism of planar 3-connected graphs. This algorithm was improved and extended by Hopcroft and Tarjan [HT74] to an $O(n \log n)$ -algorithm for the planar graph isomorphism problem (planar-GI). Then Hopcroft and Wong [HW74] showed that it is solvable in linear time. Since the constant hidden in the linear time bound is very large, the problem has been reconsidered under a more practical approach [KHC04]. The parallel complexity of planar-GI has been studied by Miller and Reif [MR91] and Ramachandran and Reif [RR94]. They showed that planar-GI \mathbf{AC}^1 -reduces to the 3-connected case and that 3-connected GI is in \mathbf{AC}^1 .

Grohe and Verbitsky [GV06] gave an alternative way to show that planar-GI is in \mathbf{AC}^1 . They proved for a class \mathcal{G} of graphs, that if every graph in \mathcal{G} is definable in a finite-variable first order logic within logarithmic quantifier depth, then the isomorphism problem for \mathcal{G}

Supported by DFG grants Scho 302/7-2 and TO 200/2-1.

is in \mathbf{AC}^1 . Later Verbitsky [Ver07] showed that planar 3-connected graphs are definable with 15 variables and quantifier depth $O(\log n)$ which leads to a 14-dimensional Weisfeiler-Lehman algorithm. With the reduction of [MR91] one obtains a new \mathbf{AC}^1 -algorithm for planar-GI.

In the above papers on planar-GI, the authors consider first 3-connected graphs. The reason is a result due to Whitney [Whi33] that every planar 3-connected graph has precisely two embeddings on a sphere, where one embedding is the mirror image of the other. Moreover, one can efficiently compute these embeddings. Weinberg [Wei66] used these embeddings to compute a code for a graph, such that isomorphic graphs will have the same code. We call a code with this property a *canonical code* for the graph.

Some of the subroutines in the above algorithms have complexity below \mathbf{AC}^1 . Allender and Mahajan [AM00] showed that planarity testing is hard for \mathbf{L} and in symmetric logspace, \mathbf{SL} . Since $\mathbf{SL} = \mathbf{L}$ [Rei05], planarity testing is complete for logspace. Furthermore Allender and Mahajan [AM00] showed that a planar embedding can be computed in logspace. Also the connectivity structure of a (undirected) graph can be computed in logspace [NTS95]. Hence a natural question is whether planar-GI is in logspace.

While this question remains open, we considerably improve the upper bound for planar-GI for 3-connected graphs in Section 3, namely from \mathbf{AC}^1 to unambiguous logspace, in fact to $\mathbf{UL} \cap \mathbf{coUL}$. Like Weinberg, we construct codes for the given graphs. In order to use only logarithmic space, our code is constructed via a spanning tree, which depends on the planar embedding of the graph. A crucial tool in the construction of the spanning tree is based on a recent result by Bourke, Tewari, and Vinodchandran [BTV07] that the reachability problem for planar directed graphs is in $\mathbf{UL} \cap \mathbf{coUL}$. They built on work of Reinhard and Allender [RA00] and Allender, Datta, and Roy [ADR96]. We argue in Section 4 that their algorithm can be modified to not just solve reachability questions but to compute distances between nodes in $\mathbf{UL} \cap \mathbf{coUL}$.

The embedding of a planar graph can be represented as a *rotation scheme*. Intuitively this gives the edges in clockwise or counter clockwise order around each node such that it leads to a planar drawing of the graph. Rotation schemes have also been considered for non-planar graphs. We talk of *oriented graphs* in this case. We extend our results to the isomorphism problem for oriented graphs. There one has given two graphs G and H and a rotation scheme for each of the graphs. One has to decide whether there is an isomorphism between G and H that respects the rotation schemes. In Section 5 we show that the problem is in \mathbf{NL} .

With respect to lower bounds, GI is known to be hard for DET [Tor04], where DET is the class of problems that are \mathbf{NC}^1 -reducible to the determinant defined by [Coo85]. In fact, already the isomorphism problem for tournament graphs is hard for DET [Wag07]. We show in Section 6 that the isomorphism problem for planar 3-connected graphs is hard for logspace.

2. Preliminaries

Basically, \mathbf{L} and \mathbf{NL} are the classes of languages computable by a deterministic and nondeterministic logspace bounded Turing machine, respectively. A nondeterministic Turing machine is called *unambiguous*, if it has at most one accepting computation on any

input. The class of languages computable by unambiguous logspace bounded Turing machines is denoted by **UL**. **NL** is known to be closed under complement [Imm88, Sze88], but it is open for **UL**.

The functional version of **L** is denoted by **FL**. It is known that **FL**-functions are closed under composition, i.e. $\mathbf{FL} \circ \mathbf{FL} = \mathbf{FL}$. The proof goes by recomputing bits of the function value of the first function each time such a bit is needed by the second function. The same argument works when we consider functions that are computed by unambiguous logspace bounded Turing machines. If we call the class **FUL**, then this says that $\mathbf{FUL} \circ \mathbf{FUL} = \mathbf{FUL}$. We need a further property of **UL**:

Lemma 2.1. $L^{\mathbf{UL} \cap \mathbf{coUL}} = \mathbf{UL} \cap \mathbf{coUL}$.

Proof. Let M be a logspace oracle Turing machine with oracle $A \in \mathbf{UL} \cap \mathbf{coUL}$. Let M_0, M_1 be (nondeterministic) unambiguous logspace Turing machines such that $L(M_0) = \overline{A}$ and $L(M_1) = A$. An unambiguous logspace Turing machine M' for $L(M, A)$ works as follows on input x :

Simulate M on input x . If M asks an oracle question y , then nondeterministically guess whether the answer is 0 or 1.

- If the guess is answer 0, then simulate M_0 on input y . If M_0 accepts, then continue the simulation of M with oracle answer 0. If M_0 rejects then reject and halt.
- If the guess is answer 1, then simulate M_1 on input y . If M_1 accepts, then continue the simulation of M with oracle answer 1. If M_1 rejects then reject and halt.

Finally accept iff M accepts.

Note that M' is unambiguous because M_0 and M_1 are unambiguous and of the two guessed oracle answers always exactly one guess is correct. ■

Let $G = (V, E)$ be an undirected graph with vertices $V = V(G)$ and edges $E = E(G)$. Let $G - \{v\}$ denote the induced subgraph of G on $V(G) \setminus \{v\}$. The *neighbours* of $v \in V$ are $\Gamma(v) = \{u \mid (v, u) \in E\}$. By E_v we denote the edges going from v to its neighbors, $E_v = \{(v, u) \mid u \in \Gamma(v)\}$. By $d(u, v)$ we denote the distance between nodes u and v in G , which is the length of a shortest path from u to v in G .

A graph is *connected* if there is a path between any two vertices in G . A vertex $v \in V$ is an *articulation point* if $G - \{v\}$ is not connected. A pair of vertices $u, v \in V$ is a *separation pair* if $G - \{u, v\}$ is not connected. A *biconnected graph* contains no articulation points. A *3-connected graph* contains no separation pairs.

A *rotation scheme* for a graph G is a set ρ of permutations, $\rho = \{\rho_v \mid v \in V\}$, where ρ_v is a permutation on E_v that has only one cycle (which is called a rotation). Let ρ^{-1} be the set of inverse rotations, $\rho^{-1} = \{\rho_v^{-1} \mid v \in V\}$. A rotation scheme ρ describes an embedding of graph G in the plane. We call G together with ρ an *oriented graph*. If the embedding is planar, we call ρ a *planar rotation scheme*. Note that in this case ρ^{-1} is a planar rotation scheme as well. Allender and Mahajan [AM00] showed that a planar rotation scheme for a planar graph can be computed in logspace.

If a planar graph is in addition 3-connected, then there exist precisely two planar rotation schemes [Whi33], namely some planar rotation scheme ρ and its inverse ρ^{-1} . This is a crucial property in our isomorphism test.

3. Planar 3-Connected Graph Isomorphism

In this section we prove the following theorem.

Theorem 3.1. *The isomorphism problem for planar, 3-connected graphs is in $\mathbf{UL} \cap \mathbf{coUL}$.*

In 1966, Weinberg [Wei66] presented an $O(n^2)$ algorithm for testing isomorphism of planar 3-connected graphs. The algorithm computes a *canonical form* for each of the two graphs. This is a coding of graphs such that these codings are equal iff the two graphs are isomorphic. For a 3-connected graph G , the algorithm starts by constructing a code for every edge of G and any of the two rotation schemes. Of all these codes, the lexicographical smallest one is the code for G .

For a designated edge (s, t) and a rotation scheme ρ for G , the code is constructed roughly as follows. Every undirected edge is considered as two directed edges. Now one can define an Euler tour based on some rules for selecting the next edge. Basically, the rules distinguish between the case whether a vertex or edge was already visited or not. The next edge to consider is chosen to the left or right of the active edge according to ρ . Define edge (s, t) to be the start of the tour. The code consists of the nodes as they appear on the tour, where the names are replaced by the order of their first appearance on the tour. That is, the code starts with $(1, 2)$ for the edge (s, t) and every later occurrence of s or t on the tour is replaced by 1 or 2, respectively.

Weinberg's algorithm doesn't work in logspace, because one has to store the vertices and edges already visited. We show how to construct a different code in \mathbf{UL} . Let (s, t) be a designated edge and ρ be a rotation scheme for G . Our construction makes three steps.

- (1) First we compute a canonical spanning tree T for G . This is a spanning tree which depends on (s, t) , ρ , and G , but not on the way these inputs are represented.
- (2) Next we construct a canonical list L of all edges of G . To do so, we traverse T and enumerate the edges of T and their neighbor edges according to ρ . The list L does not depend on the representation of G , ρ or T .
- (3) Finally we rename the vertices depending on the position of their first occurrence in the list L and get a code word for G with respect to (s, t) and ρ .

We will see that the spanning tree in step 1 can be computed in (the functional version of) $\mathbf{UL} \cap \mathbf{coUL}$. The list and the renaming in step 2 and step 3 can be computed in logspace, \mathbf{L} . Therefore the composition of the three steps is in $\mathbf{UL} \cap \mathbf{coUL}$.

The overall algorithm has to decide whether two given graphs G and H are isomorphic. To do so we fix (s, t) and ρ for G and cycle through all edges of H as designated edge and the two possible permutation schemes of H . Then G and H are isomorphic iff we find a code for H that matches the code for G . It is not hard to see that this outer loop is in logspace. Therefore the isomorphism test stays in $\mathbf{UL} \cap \mathbf{coUL}$.

Step 1: Construction of a canonical spanning tree

We show that the following problem can be solved in unambiguous logspace.

- *Input:* An undirected graph $G = (V, E)$, a rotation scheme ρ for G , and a designated edge $(s, t) \in E$.
- *Output:* A canonical spanning tree $T \subseteq E$ of G .

Recall that by a canonical spanning tree we mean that T does not depend on the input representation of ρ or G , any representation will result in the same spanning tree T .

The idea to construct the spanning tree is to traverse G with a breath-first search starting at node s . The neighbors of a node are visited in the order given by the rotation scheme ρ . Since the algorithm should work in logspace, we cannot afford to store all the nodes that we already visited, as in a standard breath-first search. We get around this problem by working with distances between nodes.

We start with the nodes at distance 1 from s . That is, write (s, v) on the output tape, for all $v \in \Gamma(s)$. Now let $d \geq 2$ and assume that we have already constructed T up to nodes at distance $\leq d - 1$ to s . Then we consider the nodes at distance d from s . Let w be a node with $d(s, w) = d$. We have to connect w to the tree constructed so far. We do so by computing a shortest path from s to w . Ambiguities are resolved by using the first feasible edge according to ρ . We start with (s, t) as the active edge (u, v) .

- If $d(u, w) > d(v, w)$, then (u, v) is the first edge encountered that is on a shortest path from u to w . Therefore we go from u to v and start searching the next edge from v . As starting edge we take the successor of (v, u) . That is, $\rho_v(v, u)$ is the new active edge.
- If $d(u, w) \leq d(v, w)$, then (u, v) is not on a shortest path from u to w . Then we proceed with $\rho_u(u, v)$ as the new active edge.

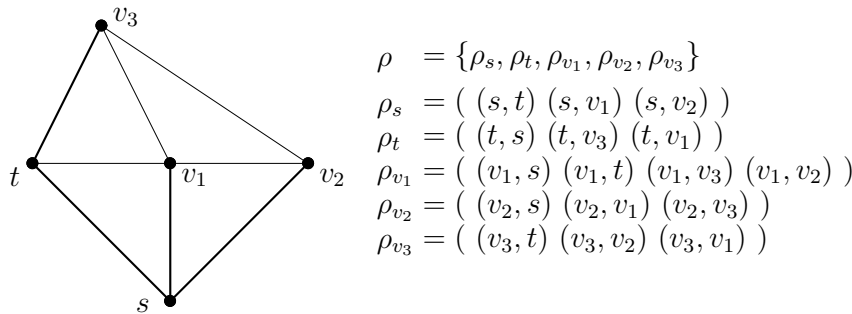
After $d - 1$ steps in direction of w the node v of the active edge (u, v) is a predecessor of w on a shortest path from s to w . Then we write (v, w) on the output tape. The following pseudo-code summarizes the algorithm.

```

for all  $v \in \Gamma(s)$  do output  $(s, v)$ 
for  $d \leftarrow 2$  to  $n - 1$  do
  for all  $w \in V$  such that  $d(s, w) = d$  do
     $(u, v) \leftarrow (s, t)$ 
    for  $k \leftarrow 1$  to  $d - 1$  do
      while  $d(u, w) \leq d(v, w)$  do  $(u, v) \leftarrow \rho_u(u, v)$ 
       $(u, v) \leftarrow \rho_v(v, u)$ 
    output  $(v, w)$ 

```

The spanning tree T is canonical because its construction depends only on ρ , edge (s, t) , and edge set E . The following figure shows an example of a spanning tree T for a graph G with rotation function ρ which arranges the edges in clockwise order around each vertex.



Except for the computation of the distances, the algorithm works in logspace. We have to store the values of d , k , u and v , and the position of w , plus some extra space for doing calculations. We show in Theorem 4.1 below that the distances can be computed in $\mathbf{UL} \cap \mathbf{coUL}$. By Lemma 2.1 the canonical spanning tree can be computed in $\mathbf{UL} \cap \mathbf{coUL}$.

Step 2: Computation of a canonical list of all edges

We show that the following problem can be solved in logspace.

- *Input:* An undirected graph $G = (V, E)$, a rotation scheme ρ for G , a spanning tree $T \subseteq E$ of G , and a designated edge $(s, t) \in T$.
- *Output:* A canonical list L of all edges in E .

Recall that by a canonical list we mean that the order of the edges as they appear in L does not depend on the input representation of ρ , G or T , any representation will result in the same list.

The idea is to traverse the spanning tree in a depth-first manner. At each vertex visit all incident edges in breath-first manner according to ρ until the next edge contained in the spanning tree is reached.

We start the traversal with edge (s, t) as the active edge (u, v) . We write (u, v) on the output tape and then compute the next active edge as follows:

- If $(u, v) \in T$ then we walk depth-first in T from u to v , consider the edge (v, u) and take its successor according to ρ_v , i.e., $\rho_v(v, u)$ is the new active edge.
- If $(u, v) \notin T$ then we proceed breath-first with $\rho_u(u, v)$ as the new active edge.

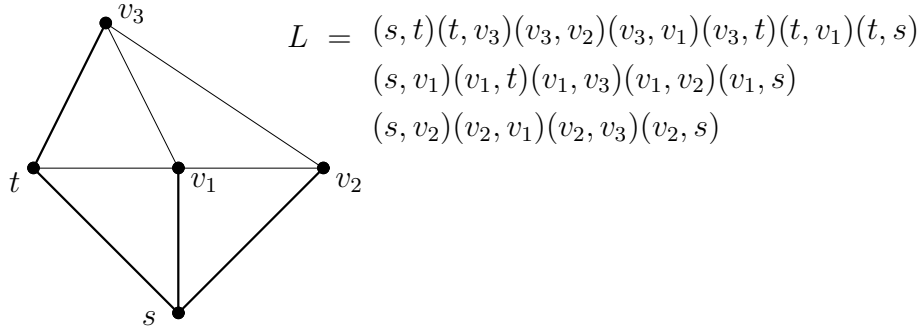
This step is repeated until the active edge is again (s, t) . Then we have traversed all edges in E . Every undirected edge is encountered exactly twice, once in each direction. The following pseudo-code summarizes the algorithm.

```

 $(u, v) \leftarrow (s, t)$ 
repeat
  output  $(u, v)$ 
  if  $(u, v) \in T$  then  $(u, v) \leftarrow \rho_v(v, u)$ 
  else  $(u, v) \leftarrow \rho_u(u, v)$ 
until  $(u, v) = (s, t)$ 

```

Clearly, the algorithm works in logspace. The list L is canonical because its construction depends only on ρ , edge (s, t) , and sets E and T . Since T is canonical as well, L depends actually only on ρ , (s, t) , and E . The following figure shows an example for L .



Step 3: Renaming the vertices

The last step is to rename the vertices in the list L such that they become independent of the names they have in G . This is achieved as follows: consider the first occurrence (from left) of node v in L . Let $k - 1$ be the number of pairwise different nodes to the left of v . Then all occurrences of v are replaced by k . Recall that L starts with the edge (s, t) . Hence

all occurrences of s get replaced by 1, all occurrences of t get replaced by 2, and so on. Call the new list $code(G, \rho, s, t)$.

Given L as input, the list $code(G, \rho, s, t)$ can be computed in logspace. We start with the first node v (which is s) and a counter k , that counts the number of different nodes we have seen so far. In the beginning, we set $k = 1$.

- If v occurs for the first time, then we output k and increase k by 1.
- If v occurs already to the left of the current position, then we have to determine the number, v got at its first occurrence. To do so, we determine the first occurrence of v and then count the number of different nodes to the left of v at its first occurrence.

It is not hard to see that this can be done in logspace.

Then we go to the next node in L . Consider the example from above. The code constructed from list L for G is as follows.

$$\begin{array}{lllllll}
 L = & (s, t) & (t, v_3) & (v_3, v_2) & (v_3, v_1) & (v_3, t) & (t, v_1) & (t, s) \\
 code(G, \rho, s, t) = & (1, 2) & (2, 3) & (3, 4) & (3, 5) & (3, 2) & (2, 5) & (2, 1) \\
 \text{sequel of } L & (s, v_1) & (v_1, t) & (v_1, v_3) & (v_1, v_2) & (v_1, s) & & \\
 \text{sequel of } code & (1, 5) & (5, 2) & (5, 3) & (5, 4) & (5, 1) & & \\
 \text{sequel of } L & (s, v_2) & (v_2, v_1) & (v_2, v_3) & (v_2, s) & & & \\
 \text{sequel of } code & (1, 4) & (4, 5) & (4, 3) & (4, 1) & & &
 \end{array}$$

The renaming algorithm works in logspace. It remains to argue that the new names of the nodes are independent of their names in G . Let H be a graph which is isomorphic to G , and let φ be an isomorphism between G and H . Note that $\rho \circ \varphi$ is a rotation scheme for H . Consider the computation of the code for graph H with rotation scheme $\rho \circ \varphi$ and designated edge $(\varphi(s), \varphi(t))$. The spanning tree computed in step 1 will be $\varphi(T)$ and the list computed in step 2 will be $\varphi(L)$. Now the above renaming procedure will give the same number to node v in L and to node $\varphi(v)$ in $\varphi(L)$. For example nodes $\varphi(s)$ and $\varphi(t)$ will get number 1 and 2, respectively. It follows that $code(G, \rho, s, t) = code(H, \rho \circ \varphi, \varphi(s), \varphi(t))$. We summarize:

Theorem 3.2. *Let G and H be connected, undirected graphs, let ρ_G be a rotation scheme for G and (s, t) be an edge in G . Then G and H are isomorphic iff there exists a rotation scheme ρ_H for H and an edge (u, v) in H such that $code(G, \rho_G, s, t) = code(H, \rho_H, u, v)$.*

This completes the proof of Theorem 3.1 except for the complexity bound on computing distances in planar graphs. This is done in the next section.

4. Computing Distances in Planar Graphs

We show that distances in planar graphs can be computed in unambiguous logspace.

Theorem 4.1. *The distance between any two vertices in a planar graph can be computed in $\mathbf{UL} \cap \mathbf{coUL}$.*

Bourke, Tewari, and Vinodchandran [BTV07] showed that the reachability problem for planar directed graphs is in $\mathbf{UL} \cap \mathbf{coUL}$. Their algorithm is essentially based on two results:

- (1) Allender, Datta, and Roy [ADR96] showed that the reachability problem for planar directed graphs can be reduced to grid graph reachability. Grid graphs are graphs

who's vertices can be identified with the grid points in a 2-dimensional grid with the edges connecting only the direct horizontal or vertical neighbors.

- (2) Reinhard and Allender [RA00] showed that the **NL**-complete reachability problem for directed graphs is in $\mathbf{UL} \cap \mathbf{coUL}$ if there is a logspace computable weight function for the edges such that for every pair of vertices u and v , if there is a path from u to v , then there is a unique minimum weight shortest path between u and v .

Bourke, Tewari, and Vinodchandran [BTV07] provide such a weight function for grid graphs. Therefore the reachability problem for planar directed graphs is in $\mathbf{UL} \cap \mathbf{coUL}$.

We modify this algorithm in order to determine distances between nodes in the given planar graph G . This is adapted from the Reinhard-Allender algorithm applied to the weighted grid graph computed from G . Here, we only describe the changes that have to be made in the cited references.

We start by considering the reduction from reachability for a planar graph G to a grid graph G_{grid} [ADR96]. The reduction from G to G_{grid} is a special combinatorial embedding that introduces only degree 2 nodes, thereby it preserves the exact number of paths between any two original vertices. Vertices in G are replaced by directed cycles and edges in G are replaced by paths such that they can be embedded into a grid. For our purpose it suffices to note that one can modify the construction and *mark the original edges of G in G_{grid}* . Hence if we consider paths in G_{grid} and count only the marked edges, we get distances in G .

The next step is to define a weight function such that shortest paths in G_{grid} with respect to marked edges are unique. Bourke, Tewari, and Vinodchandran [BTV07] defined the following weight function. For an edge e let

$$w_0(e) = \begin{cases} n^4, & \text{if } e \text{ is an east or west edge,} \\ n^4 + i, & \text{if } e \text{ is a north edge in column } i, \\ n^4 - i, & \text{if } e \text{ is a south edge in column } i. \end{cases}$$

Let p be a path in G_{grid} . The weight $w_0(p)$ is the sum of the weights of the edges on p and can be written as $a + bn^4$. Clearly, b is the number of edges on p . Also, it is easy to see that if another path p' with weight $w_0(p') = a' + b'n^4$ has the same weight as p , i.e.. $w_0(p) = w_0(p')$, then $a = a'$ and $b = b'$. This enforces that when we consider shortest paths between two nodes, these paths must have the same number of edges. The crucial part now is the value of a . Let p and p' be different simple paths connecting the same two vertices. Then Bourke, Tewari, and Vinodchandran [BTV07] showed that $a \neq a'$. It follows that the minimum weight path with respect to w_0 is always unique.

Now we modify the weight function in order to give priority to the marked edges. That is, we define

$$w(e) = \begin{cases} w_0(e) + n^8, & \text{if } e \text{ is marked,} \\ w_0(e), & \text{otherwise.} \end{cases}$$

Clearly, minimum weight paths must minimize the number of marked edges. The next parameter to minimize is the number of all edges on a path. Finally, by the same argument as above, the a -values of different simple paths that connect the same two vertices will be different. It follows that the minimum weight path with respect to w is always unique.

Reinhard and Allender [RA00] extended the counting technique of Immerman [Imm88] and Szelepcsényi [Sze88]. In addition to the number of nodes within distance k from some start node s , they also sum up the length of the shortest paths to these nodes. If the shortest paths are unique then they show that the predicate $d(s, v) \leq k$ is in $\mathbf{UL} \cap \mathbf{coUL}$. The

distance d now refers to G_{grid} because this is the input of the algorithm. By augmenting the algorithm with a counter for marked edges we also can refer to distances in G by construction of the weight function w . This suffices for our purpose because by several invocations of this procedure with different k 's we can determine $d(s, v)$ for any s and v in $\mathbf{UL} \cap \mathbf{coUL}$, where d is the distance in G .

5. Oriented Graph Isomorphism

In the previous sections we have considered planar graphs, where the planar embedding is provided by a rotation scheme. It is also interesting to consider *arbitrary* (undirected) graphs with a rotation scheme that induces some orientation, i.e. cyclic order, on the edges. In the *isomorphism problem for oriented graphs* we have given two graphs, each with a rotation scheme. One has to decide whether there is an isomorphism between the graphs that respects the orientation.

Miller and Reif [MR91] proved that the isomorphism problem for oriented graphs is in \mathbf{AC}^1 . We improve the complexity bound to \mathbf{NL} . The proof goes along the same lines as for planar-GI: compute a canonical form for each of the graphs according to the given rotation schemes such that precisely in the isomorphic case, these canonical forms are equal.

Theorem 5.1. *The oriented graph isomorphism problem is in \mathbf{NL} .*

It suffices to analyse the complexity of computing a canonical form for a graph G and a rotation scheme ρ . If G is not connected, then we determine the connected components in logspace [NTS95, Rei05] and compute canonical forms for each of them. Then we sort these canonical forms lexicographically and write them onto the output tape. Thus, we may assume that G is connected.

The three steps to compute a canonical form for a planar graph were all in logspace, except for the subroutine to compute distances, which was in $\mathbf{UL} \cap \mathbf{coUL}$. Without planarity, the best upper bound for computing distances in a graph is \mathbf{NL} : to determine if $d(u, v) \leq k$ simply guess a path of length $\leq k$ from u to v . This proves Theorem 5.1.

6. Hardness of Planar 3-Connected GI

Lindell [Lin92] proved that tree isomorphism (TI) is in \mathbf{L} . In fact, TI is complete for \mathbf{L} [JKMT03]. Since trees are planar graphs, planar-GI is hard for \mathbf{L} . We show that the problem remains hard for \mathbf{L} even when restricted to planar 3-connected graphs. All the hardness and completeness results in this section are with respect to \mathbf{AC}^0 -many-one reductions.

Theorem 6.1. *Planar 3-connected graph isomorphism is hard for \mathbf{L} .*

We reduce from the known \mathbf{L} -complete problem ORD which is defined as follows.

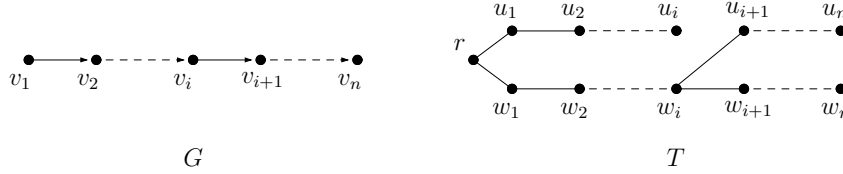
Order between Vertices (ORD)

Input: a directed graph $G = (V, E)$ that is a line, and $s, t \in V$.

Decide whether $s < t$ in the total order induced on V by G .

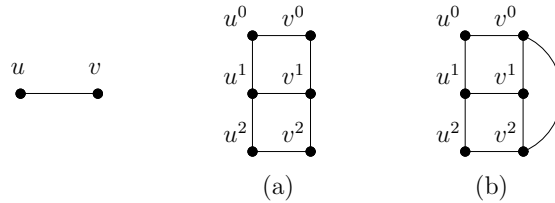
We first describe the reduction from ORD to TI [JKMT03]. Let v_1, \dots, v_n be the nodes of G in the order they appear on the line in G . In particular, v_1 is the unique node with in-degree 0 and v_n is the unique node with out-degree 0. Let $s = v_i$ and $t = v_j$. W.l.o.g. assume that $i \neq n$ (otherwise map the instance to a non-isomorphic pair of trees). The

(undirected) tree T constructed from G has two copies u_1, \dots, u_n and w_1, \dots, w_n of the line of G , and there is an additional node r that is connected to u_1 and w_1 . Up to this point, we have constructed one long line. Now the trick is to interrupt this line: take out the edge (u_i, u_{i+1}) and instead put the edge (w_i, u_{i+1}) . Let T be the resulting tree.

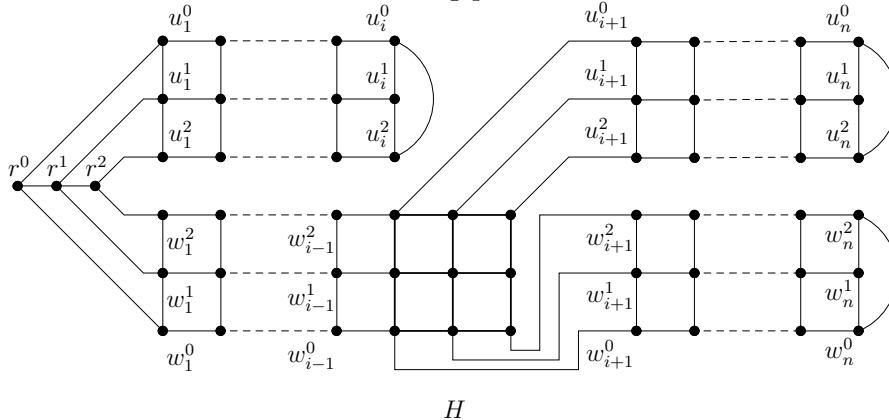


Note that there is a unique non-trivial automorphism for T : exchange u_{i+1} and w_{i+1} , \dots , u_n and w_n , and map the other vertices onto themselves. We construct two trees T_1 and T_2 from T . With respect to T , tree T_1 has two extra nodes x_0, x_1 which are connected with node u_j , and T_2 has extra nodes y_0, y_1 which are connected with node w_j . The extra edges enforce that an isomorphism between T_1 and T_2 has to map u_j to w_j , because these are the only nodes of degree 4 (for $j < n$). Now, if $v_i < v_j$, then the above automorphism of T yields an isomorphism between T_1 and T_2 . On the other hand, if $v_i \geq v_j$, then there is no isomorphism between T_1 and T_2 .

We modify T to a graph H that is no longer a tree, but planar and 3-connected. Split each node v of degree 1 or 2 in T into three nodes v^0, v^1, v^2 . Connect these nodes via edges (v^0, v^1) and (v^1, v^2) . If v has degree 1, then additionally put the edge (v^0, v^2) . Now, if (u, v) is an edge in T , where u and v have degree 1 or 2, then we have edges (u^0, v^0) , (u^1, v^1) , and (u^2, v^2) in H . The following picture illustrates the situation. In (a), node v has degree 2, in (b), node v has degree 1.



A special case is node w_i which has degree 3. For w_i we need a gadget with 9 nodes which are connected as a 3×3 grid. The connections from this graph gadget (bold lines) to the other nodes are shown in the following picture.



Now it suffices again to mark the nodes corresponding to v_j . That is, define graph H_1 as graph H plus the edge (u_j^0, u_j^2) , and H_2 as H plus the edge (w_j^0, w_j^2) . Note that H_1 and H_2 are planar and 3-connected. Furthermore, any isomorphism between H_1 and H_2 has to map u_j^0 to w_j^0 , u_j^1 to w_j^1 , and u_j^2 to w_j^2 . Again, this is only possible iff $v_i < v_j$. This completes the proof of Theorem 6.1.

A final observation is about oriented trees. An *oriented tree* is a tree with a planar rotation scheme. It is not hard to see that one can adapt Lindell's algorithm to work for oriented trees, so that the corresponding isomorphism problem is in \mathbb{L} . We show that it is also hard for \mathbb{L} .

Theorem 6.2. *Oriented tree isomorphism is complete for \mathbb{L} .*

We reduce ORD to the oriented tree isomorphism problem. Let G be the given line graph and consider again the trees T_1 and T_2 from above constructed from G in the proof of Theorem 6.1. For nodes of degree 1 or 2 there is only one rotation scheme. Therefore we only have to take care of the nodes of degree 3 and 4, i.e. w_i , w_j , and u_j .

- The rotation scheme for w_i is easy to handle: output the edges around w_i for T_1 in an arbitrary order, and choose the opposite order for w_i in T_2 . This definition fits together with the only possible isomorphism that should exchange u_{i+1} and w_{i+1} .
- In the rotation scheme for w_j the order of edges to the neighbors can be chosen as w_{j-1} , y_0 , w_{j+1} , y_1 , and around u_j in order u_{j-1} , x_0 , u_{j+1} , x_1 . Because of the symmetry of the parts (u_j, x_0) and (u_j, x_1) in T_1 and of (w_j, y_0) and (w_j, y_1) in T_2 an isomorphism mapping w_j to u_j can be defined respecting the rotation schemes for these nodes.

Now the same argument as for Theorem 6.1 shows that the oriented trees T_1 and T_2 are isomorphic iff $v_i < v_j$. This proves the theorem.

Open Problems

The most challenging task is to close the gap between \mathbb{L} and $\mathbf{UL} \cap \mathbf{coUL}$ for the planar 3-connected graph isomorphism problem. Another goal is to extend the isomorphism test to arbitrary planar graphs. If the graph is not connected, we can compute the connected components and consider them separately. Hence, we may assume that the graph is connected. Then we can determine the articulation points and the separating pairs and get the 1- and 2-connected components of the graph. For sequential algorithms to compute a canonical form for these graphs see for example [KHC04]. Miller and Reif [MR91] provide an \mathbf{AC}^1 -reduction from planar graphs to planar 3-connected graphs. We ask whether one can compute a canonical form for planar graphs in (unambiguous) logspace.

Acknowledgment

We thank Jacobo Torán and the anonymous referees for helpful comments on the manuscript.

References

- [ADR96] E. Allender, S. Datta, and S. Roy. The directed planar reachability problem. In *Proc. 16th FST&TCS*, Lect. Notes in Comp. Science 1180, pp. 322–334, Springer, 1996.
- [AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass, 1974.
- [AM00] E. Allender and M. Mahajan. The complexity of planarity testing. In *Proc. 17th STACS*, Lect. Notes in Comp. Science 1770, pp. 87–98, Springer, 2000.
- [BTV07] C. Bourke, R. Tewari, and N.V. Vinodchandran. Directed planar reachability is in unambiguous logspace. In *22nd Annual IEEE Conference on Computational Complexity*, pp. 217–221, 2007.
- [Coo85] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.
- [GV06] M. Grohe and O. Verbitsky. Testing graph isomorphism in parallel by playing a game. In *Proc. 33rd ICALP*, Lect. Notes in Comp. Science 4051, pp. 3–14, Springer, 2006.
- [HT74] J.E. Hopcroft and R.E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21, 1974.
- [HW74] J.E. Hopcroft and J.K. Wong. Linear time algorithm for isomorphism of planar graphs. In *6th ACM Symposium on Theory of Computing (STOC)*, 1974.
- [Imm88] N. Immerman. Nondeterministic space is closed under complement. *SIAM Journal on Computing*, 17:935–938, 1988.
- [JKMT03] B. Jenner, J. Köbler, P. McKenzie, and J. Torán. Completeness results for graph isomorphism. *Journal of Computer and System Sciences*, 66:549–566, 2003.
- [KHC04] J.P. Kukluk, L.B. Holder, and D.J. Cook. Algorithm and experiments in testing planar graphs for isomorphism. *Journal of Graph Algorithms and Applications*, 8(2), 2004.
- [Lin92] S. Lindell. A logspace algorithm for tree canonization (extended abstract). In *34th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 400–404. ACM, 1992.
- [Luk82] E.M. Luks. Isomorphism of graphs of bounded valance can be tested in polynomial time. *Journal of Computer and System Sciences*, 25, 1982.
- [MR91] G.L. Miller and J.H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM Journal on Computing*, 20, 1991.
- [NTS95] N. Nisan and A. Ta-Shma. Symmetric logspace is closed under complement. *Chicago Journal of Theoretical Computer Science*, 1995(Article 1), 1995.
- [RA00] K. Reinhardt and E. Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29(4):1118–1131, 2000.
- [Rei05] O. Reingold. Undirected ST-connectivity in log-space. In ACM, editor, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 376–385. ACM Press, 2005.
- [RR94] V. Ramachandran and J.H. Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49, 1994.
- [Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.
- [Tor04] J. Torán. On the hardness of graph isomorphism. *SIAM Journal on Computing (SICOMP)*, 33(5):1093–1108, 2004.
- [Ver07] O. Verbitsky. Planar graphs: Logical complexity and parallel isomorphism tests. In *24th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pp. 682–693, 2007.
- [Wag07] F. Wagner. Hardness results for tournament isomorphism and automorphism. In *Proc. 32nd MFCS*, Lect. Notes in Comp. Science 4708, pp. 572–583, Springer, 2007.
- [Wei66] H. Weinberg. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *Circuit Theory*, 13:142–148, 1966.
- [Whi33] H. Whitney. A set of topological invariants for graphs. *Amer. J. Mathematics*, 55:321–235, 1933.